COMP6700/2140 Stream Pipelines

Alexei B Khorev

Research School of Computer Science, ANU

April 2017

Topics

1. Idioms of Stream Processing
   - `filter`
   - `map`
   - `reduce`
2. `reduce` vs `collect`
3. Intermediate and Terminal Stream Operations
   - non-interference
   - stateless behaviour
   - side-effects
   - mutable reduction
4. Lazness
5. Parallel Streams
6. Re-streaming?
7. Breaking out

Idioms of streaming data processing

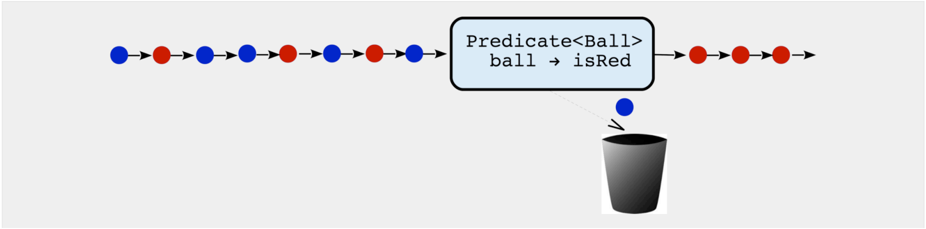
We can speak of streams and collections interchangeably since they can be transformed one into another, and can be thought as different representations of the same data set, one is *eager*, and another — *lazy*.

There are three *idioms* — typical operations which are performed on big streaming/collection data. They all have one feature in common: perform manipulation ("run code") on each element of the collection/stream in turn:

1. **Filter** — create a new stream (collection) which retains (or removes) only those elements which satisfy a certain condition
2. **Map** — transform every stream item (element of a collection) into a new one (including a different type) for further processing
3. **Reduce** — build up an aggregated value for a property of the whole data set, like sum or average, or something more complex.

The three idioms can be applied multiple times (repeated filtering, compound transformations), in variety of combinations (filtering followed by map followed by another map...). A data stream which initiates from a source, goes through one or many operation stages and ends when a *terminal operation* is executed, forms a *stream pipeline*. Stream pipelines is data processing technique which is alternative to an external iterator-based traversals.

Filter



```
List<Ball> listOfBalls = ...;
listOfBalls.stream()
           .filter(b::isRed)
           .filter(b -> b.madeOf() == MARBLE)) // can repeat filtering
         //.filter(b -> b.isRed() && b.madeOf() == MARBLE)) // or compose them
```

The method `java.util.stream.Stream<T>.filter` **takes**

- `java.util.function.Predicate<? super T>` object, or λ-expression `(T t) -> boolean`, or a method reference which returns a boolean value
- and **returns** a `java.util.stream.Stream<T>`

Filter Utilised

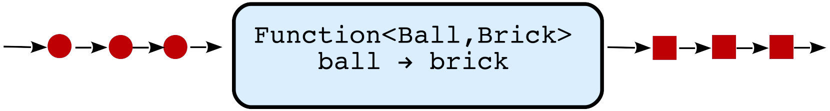
Homework 3 revisited: find factors of a given positive integer.

```
public static Stream<Integer> primes(int n) {
    return Stream.iterate(2, i -> i + 1)
                 .filter(StreamFactoring::isPrime) // or i -> isPrime(i)
                 .limit(n); // stop at n (iterate would go to infinity)
}

public static boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                    .noneMatch(i -> candidate % i == 0);
}
```

See the full code (of this short program) in StreamFactoring.java

This is still a wasteful solution since we ~~iterate~~ stream through to every new value which we check for primality repeating all but the last step which we performed at the previous step. We shall learn how to get rid of this repetitions later (lazy evaluation and memoization techniques).

Map



```
class Brick {
   final double l, w, h; // length, width, height with proportions
   Brick(Ball b) {
      ... // calculate length, width, height using size of b and proportions
   }
}
```

```
listOfBalls.stream().filter(b::isRed).map(Brick::new) // returns stream of bricks
```

The method `java.util.stream.Stream<T>.map` **takes**

- `java.util.function.Function<? super T, ? extends R>` object, or lambda-expression `(T t) -> r`, of the type `R`, or an appropriate method reference
- and **returns** a `java.util.stream.Stream<R>`

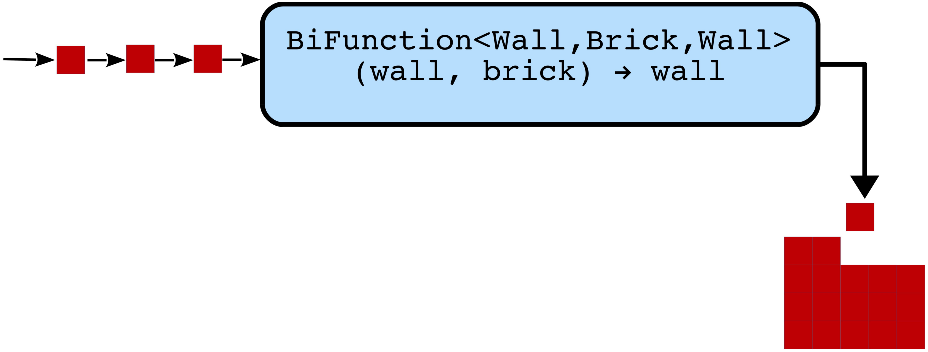
There're three “specialised” map methods: `mapToInt`, `mapToLong` and `mapToDouble` each taking an appropriate “to”-function object (`java.util.function.ToIntFunction<? super T>` and so on) to produce primitive streams (of `int`'s, `long`'s and `double`'s).

Map Utilised

Homework 4 revisited: create a list of Student objects by reading their name from a text file, and assigning them university ID's using a prescribed scheme:

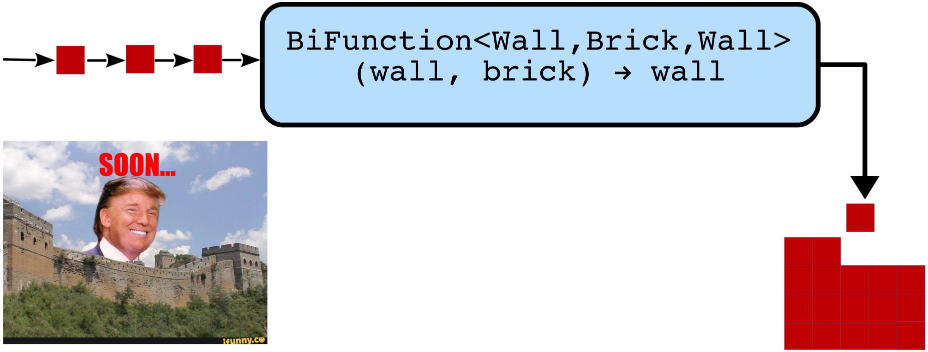
```
class Student {
   final String firstName, lastName;
   final String[] middleNames;
   public static long studentCount = 0;
   final long id;
   Student(String fullName) {
      ... some processing of fullName to break it
      ... into name components and assign corresponding name fields
      this.id = studentCount++;
   }
   ... // methods recordMarks and setGrade
}

Stream<String> fullNames = Files.lines(pathToFile);
List<Students> students = fullNames
         .map(Student::new)            // stream of student objects
         .peek(s -> s.recordMarks())   // this may query data file or user
         .peek(s -> s.setGrade())      // determine student Grade from marks
         .collect(Collectors.toList()); // collected to a list
```

Reduce



```
Wall myWall = Stream.generate(() -> new Ball(10.0)) // a stream of balls
      .filter(b::isRed)
      .map(Brick::new)
      .limit(<some-number>) // need to avoid going forever
      .reduce(new Wall(40,10),                                   // seed value
              (wall, brick) -> {wall.lay(brick); return wall;}, // accumulator
              (w1, w2) -> Wall.linkTwoWalls(w1, w2));             // combiner
```

Trump-Reduce



```
Wall myUsaMexicoWall = Stream.generate(() -> new Ball(10.0))
     .filter(b::isRed)
     .map(Brick::new)
     .limit(<some-number>) // need to avoid going forever
     .reduce(new Wall(40,10),                                   // seed value
            (wall, brick) -> {wall.lay(brick); return wall;}, // accumulator
             Wall::linkTwoWalls);                               // combiner
```

Simpleton's Reduce

If `numbers` is a collection (list, set, queue) with numbers, how to:

Sum the numbers

```
int sum = 0;
for (int x : numbers)
   sum += x;
```

Using streams

```
int sum = numbers.stream()
      .reduce(0, (x,y) -> x+y);
      //.reduce(0, Integer::sum);
```

Sum only positive numbers

```
int sum = 0;
for (int x : numbers) {
   if (x > 0) sum += x;
}
```

Using streams

```
int sum = numbers.stream()
        .filter(x -> x > 0)
        .reduce(0, (x,y) -> x+y);
```

Find number of *digits* in all positive numbers

```
int sum = 0;
for (int x : numbers) {
   if (x > 0)
      sum += x.toString().length();
}
```

Using streams

```
int sum = numbers.stream()
  .filter(x -> x > 0)
  .map(String::valueOf)
  .map(String::length)
  .reduce(0, (x,y) -> x + y);
```

Which is easier to understand? Easier to change or extend?

Astute reduce

From Assignment One in 2017: Let's build a word frequency table. We could use the same three-argument reduce method which is featured in the "Trump Wall" example, but we can have an empty table created beforehand, and use the terminal forEach method to insert every stream element in the (word frequency) table (sounds like mutable closure, right?):

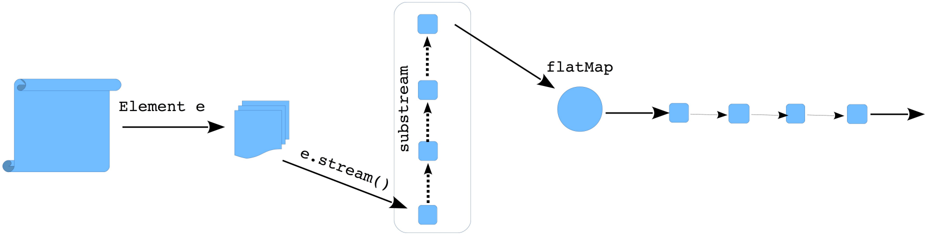
```
Map<String, Integer> freqTable = new HasMap<>(); // remove RHS if reduce used
   Files.lines(entry, StandardCharsets.ISO_8859_1) // stream of lines
     //.reduce(new HasMap<String, Integer>(),                 // seed value
     //          (m, line) -> {processLine(m, line); return m;}, // accumulator
     //          (m1,m2) -> {m1.putAll(m2); return m1;});        // combiner
     .forEach(line -> processLine(freqTable, line)); // simpler alternative

void processLine(Map<String, Integer> m, String line) {
   Stream.of(line.split("[\\P{L}]+")) //split against non-letters
         .map(String::toLowerCase)
         .forEach(word -> m.put(word, m.getOrDefault(word, 0) + 1););
}
```

We used a (relatively new) method java.util.Map.getOrDefault which acts the same as get when a key word is already present in a map (it returns the corresponding value), but when the key is missing, it returns the value given by the second argument. How short is the solution of a large part of Assignment One, no?

Astute ~~reduce~~ collect

And this is how it's done when mutability is treated as friend:

```
Map<String, Integer> freqTable =
   Files.lines(entry, StandardCharsets.ISO_8859_1) // stream of lines
     // split the line into words, turn the array into stream, normalise
     .flatMap(line -> Stream.of(line.split("\\s+")).map(Word::normalise))
     .collect(Collectors.toMap(
           Word::toString, // extract content of the word
           w -> 1,         // just another count of a given word
           (c1, c2) -> c1 + c2));
```



Collecting Results of Stream Processing

reduce methods may be not the best way to obtain results of stream data processing. Often collect and its variant provide a better alternative, or a supplementary treatment. The main consideration (apart from usability of the Stream API for a particular problem) when choosing reduce or collect is the mutability of the object being created:

- canonical application of reduce is when an immutable value is being constructed; Java has mutability, and reduce can be used to build-up a mutable aggregated object element-by-element — sometimes it looks unnatural:

```
Stream<Integer> stream = Arrays.asList(1, 2, 3, 4, 5, 6).stream();
List<Integer> numbers = stream.reduce(
           new ArrayList<Integer>(), // initialising the aggregate
           (List<Integer> l, Integer e) ->  // procedure to build it
                 { l.add(e); return l; },   // add element, return handle
           (List<Integer> l1, List<Integer> l2) -> { // how to merge
                  l1.addAll(l2);    // two parts of the aggregate if they
                  return l1; });    // were built in parallel streams
```

- collect methods are added to take advantage of mutability:

```
List<Integer> numbers = stream.collect(Collectors.toList());
```

The problem with reduce is not only aesthetic — it cannot correctly work in parallel since the concurrent modification of the same data structure operated by multiple threads can corrupt this data structure (one should create a new List each time which is detrimental for performance).

Collecting and Joining

Stream.collect takes advantage of Java's mutability in a "natural" way, unlike Stream.reduce which comes from the functional programming paradigm and is meant to return a new value every time, but can be "cheated", which costs some awkwardness — as the two examples from the previous slide show.

The collect example above used the API helper class java.util.stream.Collectors, which can be used to build a list, a map and a string object:

```
"orjner 12vqrf 7 bs znepu".chars()
   .filter(c -> Character.isLetter(c) || Character.isWhitespace(c))
   .map(Character::toLowerCase)
   .map(String2Chars::ceasarShift)
   .mapToObj(c -> String.valueOf((char)c))
   .collect(Collectors.joining("", "Caesar was warned: ",
       ", but he didn't heed the warning"));

static int ceasarShift(int c) {
    if (!Character.isLetter(c)) return c;
    return (c > 'z' - 13 ? c - 13 : c + 13);
}
```

Collector and Collectors

Collecting elements down a stream pipeline can be abstracted further by utilising the class `java.util.stream.Collector` which has the interface:

- `java.util.function.Supplier<A> supplier()`
- `java.util.function.BiConsumer<A, T> accumulator()`
- `java.util.function.BinaryOperator<A> combiner()`
- `java.util.function.Function<A, R> finisher()`

through which a collector object can provide all required components of the `collect` operation. To use such `collector` object in a terminal operation, the single-argument `collect` terminal method is used. How *Collector* instances are created (`java.util.stream.Collector` is an interface)?

- the default method `Collector.of(...)` makes one four instances above
- The utility class `javap java.util.stream.Collectors` has *factory methods*:
  - `Collectors.toList()` — to build a list
  - `Collectors.toSet()` — to build a set
  - `Collectors.toMap()` — to build a map (requires `keyMapper` and `valueMapper` function args)
  - `Collectors.toCollection()` — generalises the above three, takes a supplier which returns a new collection to be built
  - `Collectors.groupingBy(Function<T,K> classifier)` — returns collector to a map of `(K,T)` according to `classifier` (elements `T` of the stream are grouped by values to which the classifier maps them; values are used as keys, all stream elements which are mapped to the same value are collected under the key given by the value)

Composing Collectors

How can an encapsulated collector be useful? The benefit of such approach is the ability to compose multiple collectors. This example is from Java Tutorial (assuming that people is a collection of *Person* objects with attributes like gender, declared as enum type, name and age):

```
Map<Person.Sex, List<String>> namesByGender = people.stream()
   .collect(                      // cascading collecting begins
      Collectors.groupingBy(      // collector to build a map
         Person::getGender,       // classifier Person->Gender
         Collectors.mapping(      // values will be maps Name->List<Person>
            Person::getName,
            Collectors.toList()))));// the innermost collecting
```

Apart from (nested) containers (map with list values), collectors are used to perform popular "on-the-fly" processing (like statistics) and printable representation:

- `Double averageAge = people.stream()`
  `          .collect(Collectors.averagingInt(p -> p.age));`
- `IntSummaryStatistics ageSummary = people.stream()`
  `          .collect(Collectors.summarizingInt(p -> p.age));`
- `Collectors.joining(delimiter, prefix, suffix)`

  — returns a string representation of the collected aggregate (see an example two slides back, "Collecting and joining")

Intermediate and Terminal Operations

All stream operations which constitute a pipeline come in two flavours:

- **Intermediate operations** — they return a new stream, and they are *lazy*, which means that the operation is not actually carried out "right where it is present" on a pipeline, but rather that its execution is postponed until a *terminal operation* which happens *last* on the pipeline starts executing; lazy operations only define what *will* happen to stream elements (kind of a composed function definition), but the onset of these computations is delayed util a terminal operation. Laziness has big performance benefits since it reduces (or minimisers greatly) an intermediate state, and avoids examining data when it is not necessary. Examples:
  - `filter`
  - `map` and `flatMap`
  - `peek`
  - `sorted` and `distinct`
- **Terminal operations** — they traverse the stream to produce a result or a side-effect; once the terminal operation is performed, the stream is *consumed* (you need to return to the original data source to obtain, "reissue", your stream to run another pipeline through it). Almost all terminal ops are *eager* — they do perform the data traversal and process the stream elements before returning. Examples:
  - `forEach`
  - `reduce`
  - `collect`
  - `anyMatch` (and other `"match"` ops), `findFirst` and `findAny`
  - `count`

Interference with Stream Source

A stream pipeline can be *interfered* with if a behavioural parameter — one passed to a stream operation (`map` etc) — modifies (or causes to be modified) the stream's data source. Such modification of a stream source during the pipeline execution may cause exceptions, erroneous results, or unstable behaviour (when results change every time the pipeline is executed).

Interference should be avoided, but it is not always dangerous: the source can be modified without harmful effects if such modification takes place before the pipeline terminal operation starts executing:

```java
List<String> l = new ArrayList(Arrays.asList("one", "two"));
Stream<String> sl = l.stream();
l.add("three");
String s = sl.collect(joining(" "));
```

This example (taken from the Summary Javadoc for the `java.util.stream` package in the Java SE 8 API) should results into `s` having the value `"one two three"`. It is important to understand that the actual data processing starts only when a terminal operation begins executing. All prior statements (obtaining the stream, intermediate operations) are just a lengthy (or not too lengthy) set-up procedures which determine which data, how and in what order are going to be operated upon and to what end.

Side-effects during a Pipeline

Of three way to use a stream pipeline to compute an aggregate value:

- using the `forEach` terminal method
- using the three-argument `reduce`
- using the three argument `collect`

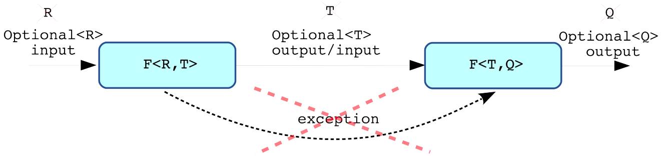
which one is more preferable?

Because the stream pipeline can be processed in sequential as well as in parallel manner, the best choice is the one which has *no side-effects* due to behavioural parameters passed to stream operations. Therefore, a `forEach`-based approach is not a good choice, `reduce` may or may not be side-effects free, and `collect` is the best option:

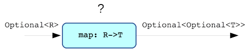
```java
List<String> results = new ArrayList<>(); // results exists beforehand
stream.filter(s -> pattern.matcher(s).matches())
      .forEach(s -> results.add(s));  // results are "side-affected"

List<String> results =
      stream.filter(s -> pattern.matcher(s).matches())
            .collect(Collectors.toList());  // No side-effects!
```

Flat Map and *Optional*

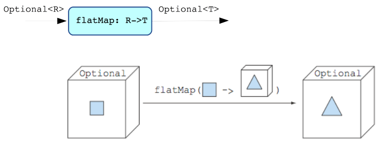
(revisiting **F2** discussion) — we need to avoid the bifurcations of *value-or-exception* in a stream pipeline operations to maintain the pipe-like ("1D") data flow. The way to achieve this is to treat the regular value and the exception as one value — *Optional*:



To avoid the morass of nested *Optional*s in a pipeline, let's realise that *Optional* can be viewed as *two-element stream*; the "optional-of-optional" may be flattened the same way as other nested streams — with `flatMap` instead of `map`:



Example: FlatteningOptional.java



Laziness

A stream pipeline which consists of only intermediate operations is *lazy* — the pipeline is not evaluated (since it's a computation defined but not executed). The example LazyStreams.java (offered by A. Shipilev in 2012) can be used to demonstrate this.

```
int[] invocations = new int[] {0,}; // on jshell scalar int would suffice
Stream<String> words = Arrays
    .asList("No one expects the Spanish Inquisition".split("\\s"))
    .stream()
    .filter(s -> {invocations[0]++; return s.length() == 3;});
Iterator<String> iter = words.iterator(); // no terminal op on words so far!
System.out.printf("%d invocations performed%n", invocations[0]);
System.out.printf("does stream have more? %s%n", iter.hasNext());
while (iter.hasNext()) {
    iter.next();
    System.out.printf("%d invocations performed%n", invocations[0]);
}
```

Before `iter.next()` is called once, no operation in the words pipeline gets executed (laziness), and once it does, it only goes for as long as the stream allows:

```
0 invocations performed
does stream have more? true
2 invocations performed
4 invocations performed
```

Stateless vs Stateful, Bounded vs Unbounded Stream Ops

A stream pipeline, depending on which operations constitute it, may or may not have state, which in turn may or may not depend on the size of the pipeline.

- `map` and `filter`

  *use every element from the input stream and produce one or zero to the output stream; they do not need to store information about stream elements which they've already processed; if the operations (λ or method references) which the user supplied to them do not have internal mutable state, `filter` and `map` remain **stateless** — they have no internal state of their own.*

- `reduce`, `count`, `sum`, `allMatch`, `anyMatch`, `noneMatch` `findFirst`, `findAny`, `max`, `min`

  *need to retain information about processed elements; yet their internal state is **bounded**, in a sense that it does not grow with the number of processed elements.*

- `sorted`, `distinct`

  *need to remember previously processed elements to do their job, so they are burdened with a state buffer which grows with the number of processed elements, their state is **unbounded**. The last two groups of stream operations are **stateful**.*

The results of computation may differ for sequential and parallel stateful pipelines (`findFirst` produces different results depending on which branch reports the finding first); care is needed in how those results are used for further computation.

External vs Internal Iterator: Burden of Relationship

(Shipilev's benchmark; reference to A. Langer stream-vs-loop study)

Sequential and Parallel Streams

Data traversal with explicit `for`-loops are *sequential*. Streams have a (designed) advantage — they can be parallelised: a pipeline can be broken into several parts and processed by different threads or processes (run on different cores). Parallelisation has to be explicitly requested:

- `balls.stream().filter(b::isRed).map(Brick::new).collect(...)` — serial pipeline
- `balls.parallelStream().filter(b::isRed).map(Brick::new).collect(...)` — a parallel pipeline; the number of parallel streams is set by the JVM option `-Djava.util.concurrent.ForkJoinPool.common.parallelism=5` (the default value varies, on a modern laptop it is usually 3)
- an existing sequential stream can be parallelised by calling the method `parallel()` (extended by the *Stream* from its parent interface `java.util.stream.BaseStream`)

Because of laziness, the sequential or parallel processing of a pipeline will be enacted only at the terminal operation, and even then the sequential or parallel orientation can be changed by calling `BaseStream.sequential()` or `BaseStream.parallel()`.

Some terminal ops are *non-deterministic* (`findFirst/findAny`), but for others, whether a stream executes sequentially or in parallel should not affect the result of the computation.

Additional work required in parallel orientation (for `reduce/collect`, this is the combine operation) and other factors (intermediate ops like `sort()` may execute sequentially even in parallel orientation of the input stream) can diminish advantages of parallelisation. As usual in concurrency matters, this is all quite subtle.

Reusing Streams

Streams cannot be reused. As soon as you call any terminal operation the stream is closed. An attempt to issue a new pipeline from a closed stream will result into the `IllegalStateException` thrown.

```
Stream<Ball> redBallStream = Stream
     .generate(() -> new Ball(10.0))
     .limit(50)
     .filter(b -> b.colour == Ball.Colour.RED);
stream.anyMatch(b -> true);     // ok
stream.noneMatch(b -> true);    // exception
```

To overcome this limitation we have to create a new stream chain for every terminal operation we want to execute, e.g. we could create a stream supplier to construct a new stream with all intermediate operations already set up:

```
Supplier<Stream<Ball>> redBallStreamSuppplier = () -> Stream
         .generate(() -> new Ball(10.0))
         .limit(50)
         .filter(b -> b.colour == Ball.Colour.RED);
redBallStreamSuppplier.get().anyMatch(s -> true);    // ok
redBallStreamSuppplier.get().noneMatch(s -> true);   // ok
```

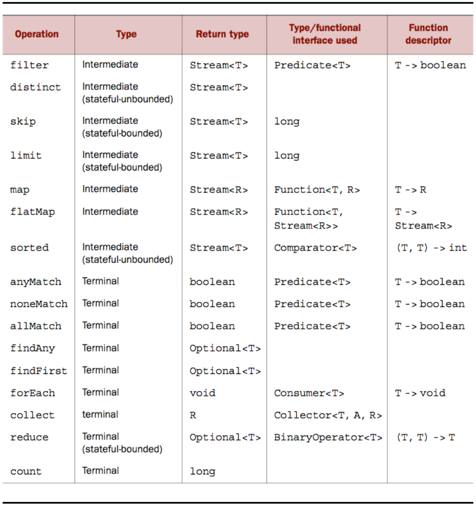
Breaking out of a Stream

How to stop an infinite pipeline when the process which the pipeline has been executing has attained its goal, *eg*, the *Trump Wall* has got built, but the stream of balls (or bricks) keeps coming?

We will discuss it after the HW6 due date...

Operations Table

The operation table (courtesy of R.G. Urma *etal* Java 8 in Action)



Operation	Type	Return type	Type/functional interface used	Function descriptor
`filter`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`distinct`	Intermediate (stateful-unbounded)	`Stream<T>`		
`skip`	Intermediate (stateful-bounded)	`Stream<T>`	`long`	
`limit`	Intermediate (stateful-bounded)	`Stream<T>`	`long`	
`map`	Intermediate	`Stream<R>`	`Function<T, R>`	`T -> R`
`flatMap`	Intermediate	`Stream<R>`	`Function<T, Stream<R>>`	`T -> Stream<R>`
`sorted`	Intermediate (stateful-unbounded)	`Stream<T>`	`Comparator<T>`	`(T, T) -> int`
`anyMatch`	Terminal	`boolean`	`Predicate<T>`	`T -> boolean`
`noneMatch`	Terminal	`boolean`	`Predicate<T>`	`T -> boolean`
`allMatch`	Terminal	`boolean`	`Predicate<T>`	`T -> boolean`
`findAny`	Terminal	`Optional<T>`		
`findFirst`	Terminal	`Optional<T>`		
`forEach`	Terminal	`void`	`Consumer<T>`	`T -> void`
`collect`	terminal	`R`	`Collector<T, A, R>`	
`reduce`	Terminal (stateful-bounded)	`Optional<T>`	`BinaryOperator<T>`	`(T, T) -> T`
`count`	Terminal	`long`		

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 8.1–8.6, 8.8, 8.9
- Oracle's Java Tutorial chapter on Aggregate Operations
- ReductionExamples from the *Oracle's Java Tutorial Code Samples*
- The `java.util.stream` package API documentation is succinct and precise (clearly, more care has been taken of writing Java docs lately ☺)
- Maurice Naftalin's Lambda FAQ has a section "Idioms and Techniques", many entries in which deal with issues of stream creation, conversion and operations (very useful)
- Maurice Naftalin's talk at JavaOne in Sept. 2016 Journey's End: Collection and Reduction in the Stream API.
- Benjamin Winterberg's Java 8 Stream Tutorial (has among others a lucid discussion of how a stream parallelisation works)